Information Sciences 573 (2021) 65–81

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Efficient access methods for very large distributed graph databases

David Luaces *, José R.R. Viqueira, José M. Cotos, Julián C. Flores

Computer Graphics and Data Engeneering Group (COGRADE), Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela (USC), Spain

ARTICLE INFO

Article history:
Received 21 July 2020
Received in revised form 15 March 2021
Accepted 20 May 2021
Available online 26 May 2021

Keywords:
Graph databases
Subgraph search
Graph query processing
Graph indexing
Subgraph isomorphism
Large scale processing

ABSTRACT

Subgraph searching is an essential problem in graph databases, but it is also challenging due to the involved subgraph isomorphism NP-Complete sub-problem. *Filter-Then-Verify* (FTV) methods mitigate performance overheads by using an index to prune out graphs that do not fit the query in a filtering stage, reducing the number of subgraph isomorphism evaluations in a subsequent verification stage. Subgraph searching has to be applied to very large databases (tens of millions of graphs) in real applications such as molecular substructure searching. Previous surveys have identified the FTV solutions GraphGrepSX (GGSX) and CT-Index as the best ones for large databases (thousands of graphs), however they cannot reach reasonable performance on very large ones (tens of millions graphs). This paper proposes a generic approach for the distributed implementation of FTV solutions. Besides, three previous methods that improve the performance of GGSX and CT-Index are adapted to be executed in clusters. The evaluation shows how the achieved solutions provide a great performance improvement (between 70% and 90% of filtering time reduction) in a centralized configuration and how they may be used to achieve efficient subgraph searching over very large databases in cluster configurations.

1. Introduction

Graph databases can be classified into two broad types. Databases of the first type consist of a single very large graph. An example of this type of databases is the semantic web. Locate all the instances of a query graph inside the database is one of the most common functionalities in many applications. This problem is known in the literature as *subgraph matching* and it is solved through the application of *subgraph isomorphism* algorithms.

Databases of the second type are composed of a very large number of small graphs. An example of this are the molecular databases, where each record stores a molecular compound, which has, amongst other properties, its molecular structure. Molecular structures are represented with small graphs composed of atoms (graph nodes) and bonds that connect them (graph edges). Some databases of this type are reaching very large sizes. On such example is the PubChem[1] database, whose size is close to one hundred million molecules. A demanded application is to retrieve from the database the entities whose molecular structure (graph) contains a given query substructure (subgraph) [1,2].

* Corresponding author at: COGRADE-CiTIUS, Rúa de Jenaro de la Fuente Domınguez, 15782, Campus Vida, Santiago de Compostela, A Coruña, Spain.
E-mail addresses: david.luaces@usc.es (D. Luaces), jrr.viqueira@usc.es (J.R.R. Viqueira), manel.cotos@usc.es (J.M. Cotos), julian.flores@usc.es (J.C. Flores).
[1] https://pubchem.ncbi.nlm.nih.gov/.

https://doi.org/10.1016/j.ins.2021.05.047
0020-0255/

The problem of finding in a database all the graphs which contain a query subgraph is a well-known problem, named *subgraph decision* problem in the literature. This problem is solved through the application of a *subgraph isomorphism* algorithm to each graph of the database. Such an straightforward solution suffers from severe performance problems when very large databases are involved, since subgraph isomorphism is a NP-Complete problem. In general, a better approach is to use a *filter-then-verify* (FTV) strategy, where the query processing task is divided into two stages. First, a *filtering stage* discards many of the graphs through the use of some indexing structure. Next, a *verification stage* tests the retrieved candidates using a *subgraph isomorphism* algorithm.

Many subgraph search solutions based on a FTV strategy have been proposed over the last years [3–11]. Among them, GraphGrepSX (GGSX) [9] and CT-Index [8] have shown the best performance when applied to databases with a large number of small graphs. Both of them use indexing structures constructed using features of the stored graphs. Specifically, GGSX encodes all the paths whose lengths are lower than a given one in a trie structure. Breadth-first scan of query and database trie structures are performed during the *filtering stage*. On the other hand, CT-Index generates a fingerprint for each stored graph, using the sub-trees and cycles whose size is lower than a maximum given one. Bitmap subset tests are performed between query and database graph fingerprints during the *filtering stage*.

In this paper, a generic approach that enables the distributed implementation of FTV subgraph searching solutions on top of large scale data processing engines is presented. Three new index structures for the filtering stage, already proposed in [12], are now adapted to be embedded in the above framework. In particular, new byte array serializations and relevant searching algorithms are implemented. An extensive evaluation of the provided solutions, both in centralized and distributed hardware architectures, was also undertaken. The main contributions are summarized as follows.

1. A generic framework for the implementation of subgraph searching approaches following a FTV strategy on top of large scale data processing engines. First, the database is divided into partitions and each partition is indexed. Resulting indexes are encoded into byte arrays and recorded into the framework distributed structures. The filtering stage performs index searching in parallel in each partition. Next, the application of the subgraph isomorphism algorithm to the retrieved candidates is also parallelized using distributed operations.
2. Three new index structures used in the filtering stage of FTV approaches, are adapted to the above distributed framework. More precisely, Bitmap GGSX (BM-GGSX) leverages the use of very large compressed bitmaps for the representation of graph references in GGSX trie nodes. Column-Wise CT-Index (CW-CTI) stores graph fingerprints in a column-wise manner, using also very large compressed bitmaps. Fingerprint subset tests are replaced by compressed bitmap intersections in the filtering stage. Finally, K-Means CT-Index (KM-CTI) organizes the stored graph fingerprints into a binary tree that is constructed through the recursive evaluation of the K-Means clustering method (using $K = 2$) over the fingerprint collection. To achieve this, effective distance and average operations were defined over fingerprints, which achieve a reduced number of bits with value 1 in the fingerprints of internal nodes of the tree, decreasing this way the probability of having to navigate through both subtrees.
3. A comprehensive performance evaluation in both centralized and distributed hardware configurations was undertaken, using databases of sizes orders of magnitude larger than previous surveys [13,14]. The new methods achieve filtering time reductions that range from 70% to 90% with respect to their already existing base GGSX and CT-Index methods, enabling filtering stage interactive approximate results in small and medium size databases (up to 1 million graphs approximately). A combination of CW-CTI for small queries and KM-CTI for large ones appears to be the best solution for query processing. The distributed implementation of the indexing structures achieves results orders of magnitude faster than the direct parallel execution of the subgraph isomporphism algorithm. It gets better performance than the centralized evaluation when either the query is small (i.e. it has low selectivity) or the database is large enough (around one million graphs). The parallel evaluation of CT-Index shows a very good performance for any type of queries when the number of available executors is large. On the other hand, the better indexing capabilities of KM-CTI makes it a good candidate to use with few executors in either small or crowded clusters.

The rest of this paper is organized as follows. Section 2 reviews related work, with special attention to GGSX and CT-Index methods. The type of graphs considered in this paper, their features and the subgraph searching problem are formalised in Section 3. The proposed generic framework for the implementation of distributed subgraph searching FTV solution is described in Section 4. Sections 5–7 describe respectively the new BM-GGSX, CW-CTI and KM-CTI index structures. The performance evaluation and its results are discussed in Section 8, and Section 9 concludes the paper and outlines lines for related further research.

2. Related work

The problem of subgraph query processing has received much attention in the data management literature. In the current state of the art this broad problem is subdivided into two major subproblems. A first one, known as *matching problem*, aims at finding all subgraph isomorphic embeddings of a query graph q in a single graph g. A second subproblem, usually named *decision problem*, has as objective the retrieval of each graph g, from a database D (multiset of graphs), which contains at least once a given query graph q.

The *matching problem* is solved through the application of subgraph isomorphism algorithms, which are usually applied to very large graphs like those of the semantic web. The Ullmann algorithm [15] provides a first backtracking-based solution to this NP-complete problem. Other initial algorithms include VF2 [16], QuickSI [17], GraphQL [18], GADDI [19] and SPath [20]. Their performance was studied in [21] using four real-world datasets of different characteristics. QuickSI had the best performance for both small and large data graphs, while GraphQL was the only algorithm to accomplish all the queries. More recent and efficient solutions include TurboIso [22] that leverages vertex similarity in the query graph, SQBC [23] that reduces the search space employing the structure of maximal cliques, BoostIso [24,25] that extends it to the stored graph and CFLMatch [26] that minimizes redundant Cartesian Products. A new recent approach [27] combines the use of a Direct Acyclic Graph (DAG) with a novel backtracking framework based on DAG-ordering. Finally, parallel approaches to this problem that define different optimization strategies to improve their performance in distributed architectures have also been implemented [28–30].

Contrary to the above, in the *decision problem* the searching process works on very large collections of small graphs. An example is the searching of molecular substructures in very large molecular databases, where the structure of each molecule is represented with a small chemical graph of atoms (vertices) and bounds (edges). The straightforward solution consists in solving the NP-complete *matching problem* between the query and each stored graph. Such an approach is clearly inefficient and even not feasible for very large databases. To reduce the search time, filter-then-verify (FTV) approaches perform a pruning phase (filter) before subgraph isomorphism is applied to the potential candidates (verify). To achieve this, indexes are constructed from graph features and used in the filter stage. To be effective, the filter stage may retrieve false positives, but it cannot discard false negatives. The false positives will be discarded by the subgraph isomorphism algorithm applied in the verify stage.

The performance of a FTV solution is mainly influenced by the indexing technique used and by subgraph isomorphism implementation chosen. Six FTV techniques [3–7,17] have been evaluated in [13]. The above survey was extended in [14] with three more algorithms whose indexes are built through the extraction of graph features, namely CT-Index [8], GRAPES [10] and GraphGrepSX [9]. This survey provides a comprehensive study of their performance and scalability and it concludes that: i) GraphGrepSX, CT-Index and GRAPES offer better performance than the previous approaches and they also scale better, ii) GRAPES fails to index large datasets due to its memory footprint, iii) the algorithms based in mining techniques (gIndex [3] and Tree+Δ [6]) are competitive only for small datasets and gCode [7] and CT-Index [8] show the best results in terms of index size.

Many of the queries of real-world scenarios have subgraph and supergraph relationships between each other. This characteristic is exploited by some approaches [31,32], through the use of caching of past queries and their results, to improve the overall performance of FTV solutions. On the other hand, straggler queries are solved in [33] by combining query rewriting with a parallel approach. A more recent work [34], combines several state-of-the-art subgraph isomorphism algorithms with GRAPES and it achieves a good performance with datasets of a large number of graphs. However, it has an extra cost in both index size and index building time. Finally, a very recent survey [35] compares three FTV solutions (GraphGrepSX, GRAPES and CT-Index), with the direct use of three advanced subgraph isomorphism solutions (GraphQL [18], CFL [26], and CFQL that combines GraphQL with CFL), and also with the combination of the indexing structures of GraphGrepSX and GRAPES with the subgraph isomorphism algorithm CFQL. Their conclusion is that the direct use of the subgraph isomorphism algorithm CFQL outperforms all the three FTV algorithms and it also achieves a performance similar to that of its combination with GraphGrepSX and GRAPES. At the same time it avoids the use of an index, with the subsequent advantages. Unfortunately, the subgraph isomorphism algorithms GraphQL [18], CFL [26] and CFQL do not use the edge labels of the graphs during the search process, thus they may not be fairly compared with the structures proposed in the present work. Besides, the largest dataset used in [35] (AIDS) to evaluate them contains only 40 k graphs, as it is also the case of previous surveys, too far from the dataset sizes used in the present work, in the order of the tens of millions of graphs.

According to the results of [14], the present work propose new indexing structures based on those of GraphGrepSX and CT-Index that may be used in new centralized and distributed FTV solutions to enable the efficient querying of very large graph datasets. The indexes may be combined with any subgraph isomorphism algorithm for the verification stage. Besides, they may also be incorporated in more complex frameworks as those described by [31–34]. For completeness, a brief description of the GraphGrepSX and CT-Index indexing structures are provided in the following two subsections.

2.1. GraphGrepSX

GraphGrepSX [9], GGSX henceforth, extracts every path with a length lower or equal to a given maximum length s from each graph of the database. The paths are sequences of vertex labels linked by edges, and they are used to incrementally build an index trie, which has a maximum depth of s. Each node of the trie records a node label and each path from the root of the trie to a given node represents an existing path in some stored graph. The nodes record also sequences of pairs (gid, r), where gid is the identifier of a stored graph and r is the number of times (repetitions) that the relevant path exists in the stored graph. Obviously, graphs whose structure do not contain the path are not recorded in the list. It is noticed that these lists of pairs increase their size linearly with the database size.

As in any FTV technique, the searching process in GGSX is divided into two stages, filtering and verification. In the filtering stage, firstly, all the paths of the query with a length lower or equal to s are generated, to build a query trie with them. Then a

joint breadth-first traversal of both query and index trie is performed that stops when all the leaf nodes of the query trie are reached. When a query leaf node is reached, a comparison between the number of repetitions r of the path recorded in the query node and the elements of the list of pairs recorded in the index trie is performed. The identifiers of the graphs that contain the path at least r times are retrieved. The lists of graph identifiers retrieved from the comparisons for each query leaf node are intersected to obtain the final candidate set of graph identifiers, which are next tested with the VF2 subgraph isomorphism algorithm in the verification stage.

GGSX was recently improved in [12] by using bitmaps in the representation of (gid, r) pairs.

2.2. CT-Index

CT-Index [8] is a FTV technique that uses as index one fingerprint for each stored graph. Each fingerprint is a bitmap of a fixed size 2^b. To generate a fingerprint, first all the features (either trees or cycles) of a maximum length s are generated from the graph. Each feature is transformed into a canonical string that is next hashed to obtain an unsigned integer n of b bits. Finally, bit number n of the fingerprint is set to one, for each generated feature. It is noticed that a fingerprint of a graph is actually a Bloom Filter [36] of all its features of maximum size s.

The filtering stage in CT-Index starts by building a fingerprint for the query graph using the same parameters (b, s) used to build the index. The query fingerprint F_q is tested for bitmap subset with each fingerprint F_g of the index ($F_q \wedge F_g = F_q$). The result set of graphs is tested for subgraph isomorphism in the verification stage using a version of the VF2 algorithm with additional heuristics.

Two new indexing structures based on CT-Index, called Column-Wise CT-Index (CW-CTI) and K-Means CT-Index (KM-CTI) were defined in [12].

3. Problem definition

The type of undirected graphs with labeled vertices and edges considered in the present work is defined as follows.

Definition 1. A *graph* is a quadruple (V, E, vl, el), where V is a set of vertices, $E \subseteq [V]^2$ is a set of undirected edges (represented as sets of two vertices) and $vl : V \to VL, el : E \to EL$ are mappings that associate a label respectively from the set of vertex labels VL and from the set of edge labels EL to each vertex and edge.

Based on the above formalism, the concept of *subgraph isomorphism* and the problem of *subgraph search*, also known as *subgraph decision problem*, are defined below.

Definition 2. If $g_a = (V_a, E_a, vl_a, el_a)$ and $g_b = (V_b, E_b, vl_b, el_b)$ are two graphs, a *subgraph isomorphism* from g_a to g_b is an injective function $f : V_a \cup E_a \to V_b \cup E_b$, such that $\forall v_a \in V_a, vl_a(v_a) = vl_b(f(v_a))$, and $\forall e_a = \{v_{a1}, v_{a2}\} \in E_a, el_a(e_a) = el_b(f(e_a))$ and $\{f(v_{a1}), f(v_{a2})\} \in E_b$. Graph g_a is said to be *subgraph isomorphic* to graph g_b.

Definition 3. If $D = [g_i]$ is a graph database (a multiset of graphs), and q is a query graph, the *subgraph searching problem* (also known as *subgraph decision problem*) aims at finding all the graphs $g_k \in D$, such that there exists at least one subgraph isomorphism f from q to g_k.

4. Generic framework for distributed subgraph searching

This section describes the design of a generic FTV subgraph searching distributed framework, and its implementation on top of the general purpose open-source cluster computing framework Apache Spark[37]. Database and index partitioning enables parallel filtering, and next candidate set re-partitioning avoids data skew during parallel verification.

4.1. Index building

The pseudocode of the generic index building procedure is given in Algorithm 1. The input data consists of an index name *iname* that enables its identification, an input graph database D and a number of partitions p. First, the number of graphs per partition (partition size *pSize*) is obtained in line 2 from the database size ($size(D)$) and the number of partitions p. Line 3 creates the base data structures for the index and records both the number of partitions and the partition size. Each iteration of the loop of lines 4–8 is executed in parallel by a different executor of the distributed processing framework. Line 5 performs a range query in the graph database D to obtain all the graphs of partition i. To achieve an efficient implementation, graphs should be partitioned and stored in a distributed data management technology, using the same auto-incremental identifier used here to perform the range query. Line 6 creates the index for the current partition i using the graphs retrieved by the above range query, and finally the generated index partition is recorded in the index structure in line 7.

Algorithm 1. Distributed Index Building

```
1: procedure BUILDINDEX(iname, D, p)
2:   pSize ← ceil(size(D)/p)
3:   idx ← createIndex(iname, p, pSize)
4:   for i ← 0, p − 1 do                ▷ Each iteration executed in parallel
5:     G ← getGraphsByRange(D, pSize ∗ i, pSize ∗ (i + 1) − 1)
6:     pIdx ← createIndexForPartition(G, pSize, i)
7:     storeIndexPartition(idx, pIdx)
8:   end for
9: end procedure
```

In the current implementation of the framework the graphs are numbered, partitioned and recorded in the document-based NoSQL database MongoDB. Apache Spark is used to implement both the distributed index building and the query processing. Thus, Spark structures and relevant operations are used as follows to implement Algorithm 1. First, line 3 of the algorithm is implemented by creating a Spark *Dataset* structure that contains one row for each partition, which records the partition number *i*. The dataset is partitioned into *p* partitions using the partition number, therefore, each partition contains just one row. The loop of lines $4-8$ is implemented by applying Spark *Map* operation to the above *Dataset*. Line 5 is implemented with a range query over the graph number on MongoDB. The index generated for each partition *pIdx* in line 6 has to be encoded in a binary format using a serialization strategy provided by the underlying indexing solution. Finally, line 7 is implemented by generating a new row in the output *Dataset* of the *Map* operation, which contains both the partition number and the serialization of the generated index structure.

4.2. Query processing

The pseudocode that shows the design of the FTV distributed subgraph searching function of the proposed framework is given in Algorithm 2. The input data consists of a query graph *q*, an input graph database *D*, stored in a distributed data management technology, and a distributed index *idx* constructed using the procedure described in the previous subsection. First, lines 2–3 respectively obtain the number of partitions *p* of the index and initialize a distributed collection for the candidate graph identifiers resulting from the filter stage. The filter stage is performed in the loop of lines 4–8, whose iterations (one for each index partition) are executed in parallel, by: i) obtaining the relevant partition from the index (line 5), ii) searching the index using the query graph (line 6) and iii) appending a new partition with the obtained candidates to the result collection *candidates*. Once the filtering stage has finished, the result candidate collection is repartitioned, to avoid data skew in the subsequent verification stage. The verification stage is performed by the loop of lines 11–22, whose iterations are also executed in parallel. First, the candidate graph identifiers of the current partition are obtained from the relevant collection in line 12. To improve the efficiency of the graph retrieval from the database, graph identifiers are ordered and folded into collections of ranges in line 13. Thus, for example, for the collection of graph identifiers $\{2, 3, 4, 6, 7, 9\}$, the following ranges are generated $\{[2-4], [6-7], [9, 9]\}$ and thus only three range queries have to be performed. Notice that a range re-partitioning strategy is required in line 9 to enable this approach. The database is queried in line 15 and for each candidate graph *g*, if the query graph *q* is *subgraph isomorphic* to *g* (line 17), then the graph is appended in line 18 to the current partition of the result output collection *result*.

Algorithm 2. Distributed Subgraph Search

```
1: function DISTRIBUTEDSUBGRAPHSEARCH(q, D, idx)
2:   p ← getNumPartitions(idx)
3:   candidates ← initializeCollection()
4:   for i ← 0, p − 1 do                ▷ Filter: Each iteration executed in parallel
5:     pIdx ← getIndexPartition(idx, i)
6:     pCandidates ← filter(q, pIdx)
7:     append(pCandidates, candidates)
8:   end for
9:   repartition(candiates, p)
10:  result ← initializeCollection()
11:  for i ← 0, p − 1 do              ▷ Verify: Each iteration executed in parallel
12:    pCandidates ← getPartition(candiates, i)
13:    pCandidateRanges ← fold(pCandidates)
14:    for all range ∈ pCandidateRanges do
```

(*continued on next page*)

```
15:        G ← getGraphsByRange(D, range.start, range.end)
16:        for all g ∈ G do
17:          if subgraphIsomorphic(q, g) then append(g, result)
18:          end if
19:        end for
20:      end for
21:    end for
22:    return result
23: end function
```

In the current implementation of the framework, based in Apache Spark and MongoDB, the filtering stage (loop of lines $4-8$) is implemented with a distributed *Map* operation over the distributed index *Dataset* structure. Each *Map* iteration searches the serialized encoding of the index stored in the *Dataset* using the query graph q. The output of the above *Map* operation is a *Dataset* of candidate graph identifiers that is re-partitioned to be used as input of the verification stage. Such a verification stage is also implemented with a *Map* operation over the candidate *Dataset* of graph identifiers.

5. Bitmap GGSX

Bitmap-GGSX (BM-GGSX for short, henceforth) [12], is an improved version of the GraphGrepSX (GGSX) index structure. BM-GGSX records the vertex and edge labels of all the paths, whose length is lower or equal than a given one, in a trie structure. The main differences with the GGSX index structure described in SubSection 2.1 are the following: i) The edge labels of the paths are stored in the BM-GGSX trie nodes, what gives additional filtering power to BM-GGSX over GGSX. ii) The lists of pairs (g, r) of graph identifiers g and repetitions r, which are stored in each GGSX node, are represented in BM-GGSX using arrays of compressed bitmaps, which makes the structure more compact and efficient.

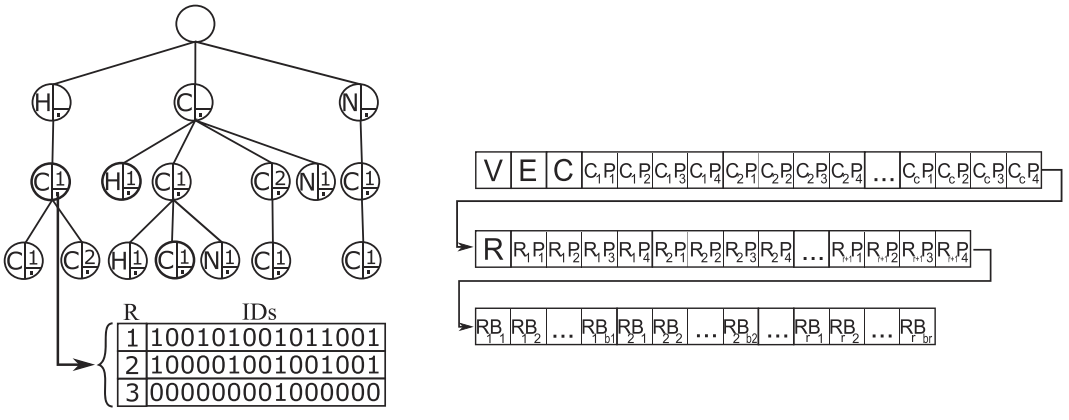
To give a precise description of the BM-GGSX data structure and relevant algorithms, let us first formalise the concept of path contained in a graph.

Definition 4. A *path* p of length $s > 0$ contained in a graph $g = (V, E, vl, el)$, denoted $p^s \subset g$, is a sequence of the form $< v_1, e_1, \ldots, v_s, e_s, v_{s+1} >$, where $\forall i \neq j, 1 \leqslant i, j \leqslant s+1, v_i \in V, v_j \in V, v_i \neq v_j$ and $\forall i \neq j, 1 \leqslant i, j \leqslant s, e_i \in E, e_j \in E, e_i \neq e_j, e_i = \{v_i, v_{i+1}\}$. Nodes $\{v_1, v_{s+1}\}$ are called the boundaries of p^s.

Let $D = [g_1, g_2, \ldots, g_n]$ be a graph database of size n. If S denotes a maximum given length, then $P(S, D)$ denotes the set of all paths whose length is lower or equal to S that are contained in some graph of D. Formally,

$$P(S, D) = \{p^s | s \leqslant S \wedge p^s \subset g \wedge g \in D\}$$

Each node v of the BM-GGSX trie structure, except the root one, records a vertex label and an edge label (which might be empty). A path in the trie from the root to a node v of level s represents the sequence of labels of at least one path p^s of length s of $P(S, D)$. It is obvious that the maximum depth of the trie is S. Let $paths(v)$ denote all those paths p^s whose sequence of labels is represented by the trie path from the root to v. If R is the maximum value for the repetitions, then node v records a two dimensional array of $R \times n$ bits, where element (i, j) is 1 if and only if graph g_j contains at least i paths of $paths(v)$. It is noticed that if the bit (i, j) is 1, then all the bits (x, j), such that $x < i$, must also be 1.



(a) BM-GGSX index trie (b) Serialization of an index trie node

Fig. 1. BM-GGSX index trie and bitmap serialization of one of its nodes.

Fig. 1(a) illustrates the BM-GGSX trie of a graph database containing molecular structures, where vertex labels represent chemical elements of atoms (C, H, O, etc.) and edge labels represent types of bounds between atoms (1 for single, 2 for double, etc.). It is noticed that all nodes of level one of the trie have a null value for the edge label, since by definition, all paths start with a vertex. The bit array is shown only for the trie node that represents all the paths with labels "H1C". Such bit array shows, for example, that only the graph g_9 contains 3 paths with those labels.

To achieve an implementation that may be embedded in the generic framework described in Section 4, the above trie structure has to be encoded and serialized into a byte array. Fig. 1(b) depicts the bytes of the serialization of a trie node. The first two bytes are used to record references to the vertex (V) and edge (E) label entries in the respective dictionaries. Byte C records the number of children of the node. Next, for each child node, four bytes $[C_iP_1 - C_iP_4]$ for child i) are used to record a pointer to its location in the byte array. Leaf nodes have $C = 0$ and children pointers are not recorded. Byte R records the number of rows in the bit array of the node, i.e., the maximum number of repetitions. For each row of the bit array, a pointer of four bytes $[R_iP_1 - R_iP_4]$ for row i) records the location of the serialization of the row in the byte array. Each row bitmap is compressed into a byte array ($R_iB_1R_iB_2 \ldots R_iB_{bi}$ for row i) using the Enhanced Word-Aligned Hybrid bitmap compression method [38], therefore each row may have a different size in its serialized representation.

Algorithm 3. BM-GGSX Searching

```
1: function SEARCH(Q, b)
2:   if isEmpty(Q) then return b
3:   else
4:     np ← dequeue(Q)
5:     if isLeafNode(getQueryNode(np)) then
6:       db ← getBitArray(getDBNode(np))
7:       r ← getRepetitions(getQueryNode(np))
8:       if r < rowsNumber(ba) then b ← b ∧ db[r]
9:       else return compressedBitArray(0, n)
10:    end if
11:    else
12:      for all qChild ∈ getChildren(getQueryNode(np)) do
13:        dbChild ← find(qChild, getChildren(getDBNode(np)))
14:        if isNull(dbChild) then return compressedBitArray(0, n)
15:        else enqueue(nodePair(qChild, dbChild), Q)
16:        end if
17:      end for
18:    end if
19:    return search(Q, b)
20:  end if
21: end function
```

A pseudocode for the search algorithm of BM-GGSX is shown in Algorithm 3. The *search* function has two parameters: i) Q is a queue structure that stores pairs of trie nodes of the form (q, db), where q is a query trie node and db is a database trie node. ii) b is a compressed bit array that records the current result of the searching process, i.e., bit i is set to 1 if graph g_i is part of the search result. The initial call to the search algorithm, which is performed by function *filter* in line 6 of Algorithm 2, has the form.

$$search((qTrieRoot, dbTrieRoot), compressedBitArray(1, n)),$$

where *qTrieRoot* and *dbTrieRoot* are respectively the root nodes of the query and database tries, and $bitarray(1, n)$ returns a compressed bit array with n 1*s* (remember that n is the number of graphs in the database). The use of a queue structure of node pairs enables the joint bread-first traversal of both query and database tries. If the queue is empty then the search process finishes returning the current result b (line 2). Otherwise, the current pair of nodes to be processed np is obtained from the queue in line 4. If the query trie node is a leaf node (line 5), then, in lines $6 - 10$, the number of repetitions r of the query trie node is used to obtain a row from the bit array db of the database trie node, which is next intersected with the current bit array b (line 8). If r is greater than the number of rows in db, it means that the current path has more repetitions in the query than in any database graph, and therefore, the search process must return with no results, i.e., a bit array of all 0*s* (line 9).

If the current query trie node is an internal node, then the children of both current query trie node and dababase trie node have to be obtained to continue with the joint bread first traversal (lines $12 - 17$). For each child of the query trie node (line 12), function *find* in line 13 tries to obtain a node with the same tags (both vertex and edge tags) from the children of current database trie node. If such a matching node does not exist then the search process finishes without results (line 14), since the

current path of the query trie is not present in the database trie. Otherwise, the node pair $(qChild, dbChild)$ is added to the queue (line 15) to be processed in a future recursive call to the *search* function. The search process continues with the next pair of the queue with the recursive call of line 19. The graphs identifiers resulting from the filtering process are obtained from the result bit array by getting the indexes of the bits that are set to 1.

6. Column-Wise CT-Index

Column-Wise CT-Index (CW-CTI in short)[12] is a modification of the CT-Index structure described in SubSection 2.2. The cycles and trees are the basic features extracted from the graphs to construct the CT-Index structure and therefore also the CW-CTI. Those concepts are formalized below to enable a precise description of the method.

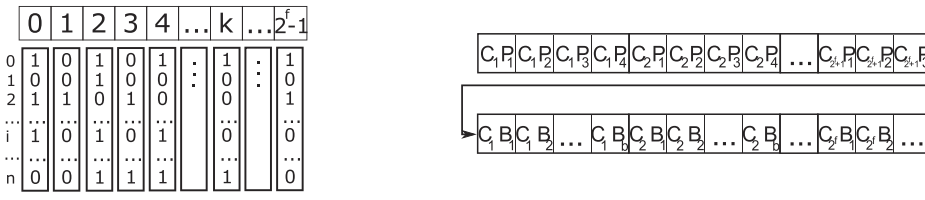
Definition 5. A *cycle* c of length $s > 0$ contained in a graph $g = (V, E, vl, el)$, denoted $c^s \subseteq g$, is a graph $c = (V_c, E_c, vl, el)$, where $V_c \subseteq V$ has the form $\{v_1, v_2, \ldots, v_s\}, E_c \subseteq E$ has the form $\{e_1, e_2, \ldots, e_s\}, \forall i, 1 \leqslant i < s, e_i = \{v_i, v_{i+1}\}$ and $e_s = \{v_s, v_1\}$.

Definition 6. A *tree* t of length $s > 0$ contained in a graph $g = (V, E, vl, el)$, denoted $t^s \subseteq g$, is a graph $t = (V_t, E_t, vl, el)$, where $\forall (v_i, v_j), i \neq j, v_i, v_j \in V_t$, it exists exactly one path p contained in g whose boundaries are $\{v_i, v_j\}$.

Algorithm 4. CW-CTI Index Building and Searching

```
1: procedure APPENDGRAPH g, idx
2:     for all c ∈ extractCycles(g, S) do
3:         idx[identifier(g)][hashCycle(c, f)] ← 1
4:     end for
5:     for all t ∈ extractTrees(g, S) do
6:         idx[identifier(g)][hashTree(t, f)] ← 1
7:     end for
8: end procedure
9: function SEARCH q, idx
10:    result ← compressedBitArray(1, n)
11:    for all c ∈ extractCycles(q, S) do
12:        result ← result ∧ idx[][hashCycle(c, f)]
13:    end for
14:    for all t ∈ extractTrees(q, S) do
15:        result ← result ∧ idx[][hashTree(t, f)]
16:    end for
17:    return result
18: end function
```

Let $D = [g_1, g_2, \ldots, g_n]$ be a graph database of size n. If f is a positive integer, then the CW-CTI index structure is a two dimensional array of $n \times 2^f$ bits. The main difference with the CT-Index structure is that the implementation of CW-CTI records the array column-wise using the Enhanced Word-Aligned Hybrid bitmap compression method [38], contrary to the row-wise recording of graph fingerprints of CT-Index. Fig. 2(a) shows this structure for a generic database. The pseudocode of the algorithm that appends a new graph g to an already existing CW-CTI index *idx* is given in procedure *appendGraph* of Algorithm 4. First, lines 2–4 process the cycles extracted from g and next lines 5–7 do the same with the extracted trees. For each extracted feature (either cycle or tree of a size lower or equal to a given one S), a function (either *hashCycle* or *hashTree*) generates first a



(a) CW-CTI Index (b) CW-CTI Byte Array Serialization

Fig. 2. Example of CW-CTI index structure and bitmap serialization.

canonical string representation of the feature using the labels of vertices and edges, and next, it applies a hash function to such cannonical representation to obtain an integer index k ($0 \leqslant k \leqslant 2^f - 1$). Using the graph identifier as a row index and k as a column index, the bit of the structure corresponding to the extracted feature is set to 1.

Fig. 2(b) illustrates the byte array of the serialized structure. The compressed bit arrays corresponding to each possible value of the hash function (fingerprint bit) are recorded one next to the other. Due to the use of compression, each bit array may be serialized into a distinct number of bytes ($C_kB_1 - C_kB_{bk}$ for array number k). Therefore, to enable direct access, one pointer of four bytes has to be recorded for each of them ($C_kP1 - C_kP_4$ for array number k).

The pseudocode of the search algorithm of CW-CTI is shown in function *search* of Algorithm 4. The result is initialized to a compressed bit array of n 1*s* (line 10). For each feature (either cycle or tree) extracted from the query graph q, the corresponding bit array is obtained from the index, using again functions *hashCycle* and *hashTree*, respectively, and it is intersected with the current result. Therefore, at the end of the execution, the result bit array has 1*s* only in the positions corresponding to graphs that may contain all the features extracted from the query. Notice that false positives may appear not only due to the use of features of a maximum size S, but also due to the potential collisions of the hash functions.

It is noticed that while in CT-Index, each stored fingerprint has to be tested against the query fingerprint, in CW-CTI, the number of logical AND operations depends on the number features extracted from the query, which is greater as the query size increases. Thus, it is expected that CW-CTI will get better performance in small queries.

7. K-Means CT-Index

The K-Means CT-Index (KM-CTI) [12] is another evolution of the CT-Index structure described in SubSection 2.2. KM-CTI uses a binary tree of fingerprints instead of the straightforward linear search approach used by CT-Index. The leaf nodes of the binary tree are the graph fingerprints. The fingerprint of each parent node is the logical disjunction of its children fingerprints. Thus, if the query test fails for a parent node it would also fail for all its children and they do not need to be tested. Fig. 3 depicts an example of a KM-CTI fingerprint binary tree for a database of 8 graphs. If the database is large enough and internal nodes have not many bits set to 1, then many branches would not need to be explored and therefore the expected number of comparisons would be lower than in the original CT-Index. To minimize the number of bits set to 1 in internal nodes, the fingerprints under the same branch must be as much similar as possible. To achieve this, the index building method of KM-CTI uses the K-Means [39,40] clustering method.

Functions *distance* and *mean* that enable the application of K-means to sets of fingerprints are defined below, but before some notation has to be introduced. If f is a positive integer, then F and F_i are used to denote *fingerprints*, i.e., one dimensional arrays of 2^f bits. Symbols $\wedge, \vee, =, \neg, \oplus$ denote respectively bitwise operations *and*, *or*, *equal*, *not* and *exclusive or* between fingerprints. If i is an integer $0 \leqslant i < 2^f$, then $F[i]$ denotes the Boolean element corresponding to bit number i of F. Finally, $int(b)$ denotes the integer element corresponding to a Boolean element b. The definition of *distance* between fingerprints is based on the well-known Hamming Distance [41]. Informally, if either of the two fingerprints is contained in the other then the distance is zero, otherwise it is the number of bits set to 1 in their exclusive or.

Definition 7. If F_1 and F_2 are two fingerprints, then $distance(F_1, F_2)$ is defined as the integer

$$d = \begin{cases} 0 & (F_1 \wedge F_2 = F_1) \vee (F_1 \wedge F_2 = F_2). \\ \sum_{i=0}^{2^f-1} int((F_1 \oplus F_2)[i]) & \text{otherwise.} \end{cases}$$

Function *mean* enables K-means algorithm to compute a fingerprint that is representative of a cluster (set) of fingerprints. Informally, bit number i of the *mean* of a set of fingerprints is set to 1 if at position i there are more fingerprints with a value of 1 than with a value of 0.

Definition 8. If $S = F_1, F_2, \ldots, F_n$ is a set of n fingerprints, then $mean(S)$ is defined as the fingerprint M such that

$$\forall i, 0 \leqslant i < 2^f, M[i] = \sum_{j=1}^{n} int(F_j[i]) \geqslant \sum_{j=1}^{n} int(\neg F_j[i])$$

Algorithm 5. KM-CTI Index Building

```
1: function KMCTITreeGF
2:     clusterFingerprints(GF, GF_L, GF_R)
3:     if empty(GF_L) ∨ empty(GF_R) then result ← leafNode(GF)
4:     else
5:         result ← internalNode(KMCTITree(GF_L), KMCTITree(GF_R))
6:     end if
7:     return result
8: end function
```

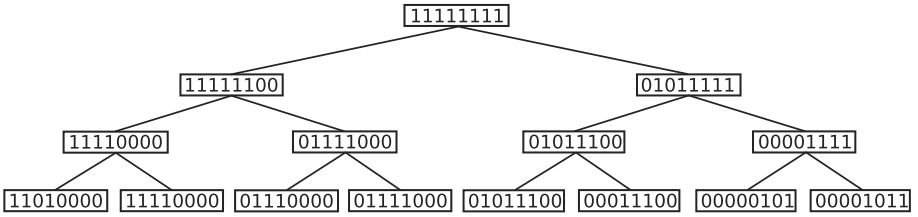


Fig. 3. Example of a KM-CTI binary tree.

Algorithm 5 shows how the KM-CTI binary tree is created from a the set of fingerprints generated for a set of graphs. The input GF is a set of pairs (g, F), where g is a graph identifier and F is its corresponding fingerprint. K-means is used (with $k = 2$) in function *clusterFingerprints* (line 2) to split the set GF of (g, F) pairs into two disjoint subsets, GF_L and GF_R. If either of the above fingerprint subsets is empty, then K-means did not manage to divide further GF and therefore a leaf node of the KM-CTI binary tree has to be created (line 3). The leaf node stores both the fingerprint computed as the disjunction of all the fingerprints in GF, and all the pairs of GF. On the other hand, if both GF_L and GF_R have elements, recursive calls to function *KMCTITree* over both subsets are performed to continue with the creation of left and right branches of the KM-CTI binary tree. An internal parent node is created in line 5 that stores pointers to left and right children and also a fingerprint computed as the disjuntion of left and right fingerprints.

KM-CTI binary tree nodes are serialized in depth-first order. The structure of the bytes of a generic internal node is illustrated in Fig. 4(a). First, b bytes $(V_1 - V_b)$ are used to record the node fingerprint (the disjunction of the fingerprints in its subtree). Next, one byte L is used to record the node type (*internal* in this case). Finally, two sequences of four bytes are used to record pointers to the left $(C_L P_1 - C_L P_4)$ and right $(C_R P_1 - C_R P_4)$ children of the node. The structure of a generic leaf node is illustrated in Fig. 4(a). The node fingerprint and the node type are recorded in the same way. Next, one byte N is used to record the number of pairs (graph identifier, fingerprint) recorded in the node. A node with two pairs is shown in the figure. For each pair i, first, four bytes $([C_i I_1 - C_i I_4])$ are used to record the graph identifier and next b bytes $(C_i V_1 - C_i V_b)$ are used to record the graph fingerprint.

Algorithm 6. KM-CTI Searching

```
1: function SEARCH F_q, idx
2:   if F_q ∧ fingerprint(idx) = F_q then
3:     if isLeaf(idx) then
4:       result ← ∅
5:       for all (g, F) ∈ graphs(idx) do
6:         if F_q ∧ F = F_q then result ← result ∪ g
7:         end if
8:       end for
9:       return result
10:    else
11:      return search(F_q, leftChild(idx)) ∪ search(F_q, rightChild(idx))
12:    end if
13:  else return ∅
14:  end if
15: end function
```

The search algorithm, whose pseudocode is give in Algorithm 6, scans the binary tree in depth-first order. A node is explored only if the query fingerprint F_q is contained in the node fingerprint (line 2). Such a test is also performed for each graph fingerprint when leaf nodes are reached (line 6).

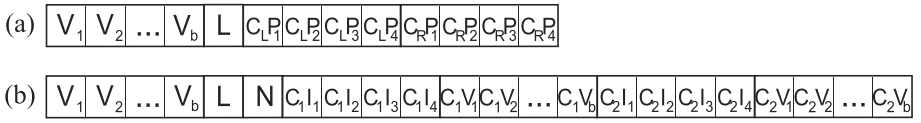


Fig. 4. KM-CTI serialization: (a) Internal node and (b) Leaf node.

8. Evaluation

The results of an extensive evaluation of both centralized and distributed implementations are discussed in this section, based on the execution of a large number of simulated queries over a very large real dataset (tens of millions of graphs). Index size and building time, and query response time (both filtering stage and total) are given for different query sizes, database sizes and different number of executors.

8.1. Evaluation framework

The hardware infrastructure consists of 16 Dell EMC PowerEdge R730 Servers, each of them with 2 Intel(R) Xeon(R) CPU E5-2630 v4 (2.20 GHz, 10 cores) processors and 384 GB RAM. All the implementations were done in Java (a jar file was obtained from CT-Index authors). Java virtual machines with 32 GB of RAM were used by all the machines in both centralized and distributed configurations. PubChem[2] was used to generate datasets of different sizes. This database contains the structure (graph) of around 97 million molecules (average of 41.40 vertices and 42.16 edges per graph). Three collections of 100 random small (8 edges), medium (20 edges) and large (40 edges) queries were generated from PubChem, following the same random walk approach of previous evaluations [8,14,12]. Random graphs are chosen from the dataset to generate random paths of the required sizes.

8.2. Evaluation of the centralized implementation

An initial evaluation of the centralized implementations was already undertaken in [12], and based on that, a maximum feature (path, subtree, cycle) length of 6 and a fingerprint size of 4096 bits was chosen. The original GGSX method was discarded due to its bad performance (BM-GGSX reduces its filtering time in around 90%). Four databases with respective sizes of 100 k, 250 k, 500 k and 1000 k graphs were extracted from PubChem. The index size and index building time for each method and database size are depicted respectively in Figs. 5(a) and 5(b). BM-GGSX has bad performance in index size, but specially bad in index building time (around one day to build the index of 1 million graphs). Regarding methods based on CT-Index, the original CT-Index and the proposed CW-CTI variant have a similar performance, whereas KM-CTI has some overhead, specially in the index size, due to the need of additional fingerprints in internal tree nodes.

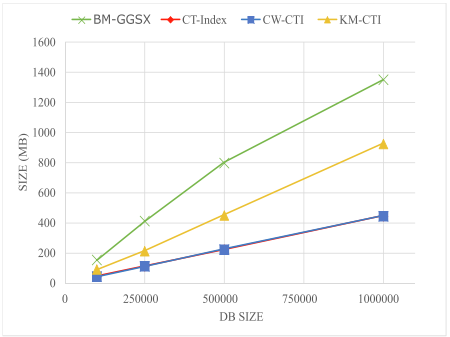
Fig. 6(a)–(c) depicts the average filtering time of each method for the generated collections of small, medium and large queries. BM-GGSX and CW-CTI show the best performance for small queries, since few features have to be searched in the index. However, when the query size increases, CW-CTI decreases its performance and KM-CTI shows its advantages. Overall, CW-CTI reaches a 85% reduction of the filtering time in small queries with respect to the original CT-Index. On the other hand, for large queries KM-CTI reduces the filtering time in around 70%-75% with respect to the original CT-Index. BM-GGSX has good performance in all the sizes, but it is reminded its bad numbers in index size and building time.

The total query time of the three proposed methods for the different query sizes is shown in Fig. 6(d)–(f). The total response time of the original CT-Index is not shown, since it is an order of magnitude higher. This is due to the slow sequential implementation of the verification stage used by CT-index and the fast parallel multi-thread implementation used by the other methods. CT-Index based methods outperform BM-GGSX in small and medium size queries due to their better filtering capabilities, in those low selective queries. As it was expected, the time of the filtering stage has only a real impact in total response time in queries with high selectivity, i.e., in large queries, where KM-CTI shows better performance than CW-CTI.

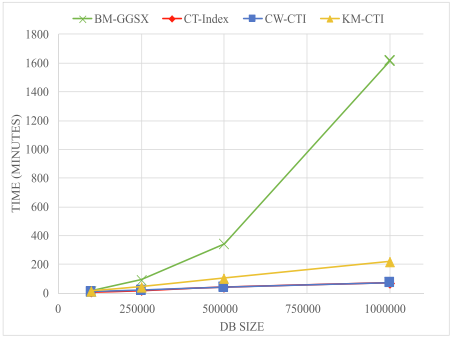
8.3. Evaluation of the distributed implementation

For the evaluation of the distributed implementations, databases with the following sizes were extracted from PubChem: 100k, 250k, 500k, 1000k, 2M, 4M, 8M, 16M, 32M, 64M and 97M. Different number of partitions were also considered for each database size, from 2^1 to 2^6, to enable the execution with as many executors as partitions. To have index building times reasonably short, the maximum number of graphs per partition considered was 500k, thus for example, the minimum number of partitions for a database of 4M graphs was 8. For databases larger then 32M only 64 partitions were used. The results obtained for the index building time are graphically shown in Fig. 7. In particular, Fig. 7(a) shows how the time required to build the index of the 4M database decreases as the number of executors increases, for each method. In line with the results obtained by the centralized implementations, it is noticed that BM-GGSX has the worse performance and KM-CTI adds some time overhead with respect to the other two CT-Index variants. A comparison of the index building time using 64 executors for different database sizes is shown in Fig. 7(b). Database sizes higher than 8M were not supported by BM-GGSX due to memory consumption. Similarly, KM-CTI could not manage sizes larger than 32M and only the distributed version of the original CT-Index supported all the database sizes. CT-Index and CW-CTI show a similar behaviour, though CT-Index performs better in very large databases (above 16M graphs). On the other hand, the performance of KM-CTI degenerates more with the database size, due to the need of recursive evaluations of the K-Means clustering method. In any case, the distributed implementation enables a great reduction in the index building time for all the methods, as it is illustrated in

[2] **https://pubchem.ncbi.nlm.nih.gov/**

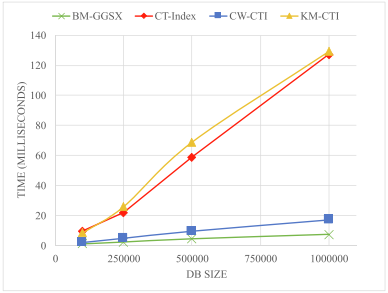


(a) Index Size

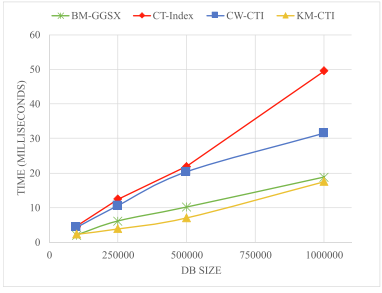


(b) Index Building Time

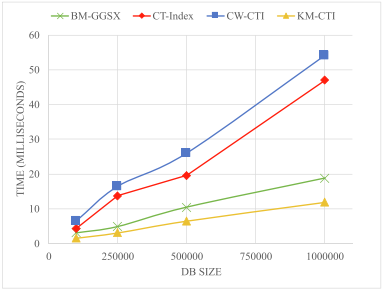
Fig. 5. Centralized index size and building time.



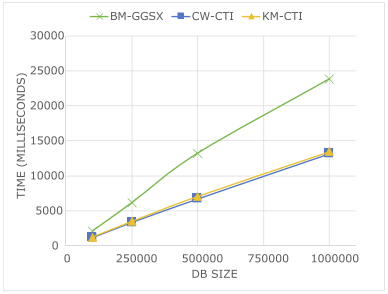
(a) 8 Edges Query Filtering Time



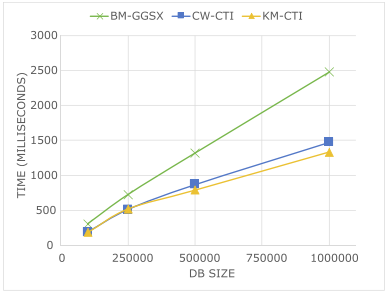
(b) 20 Edges Query Filtering Time



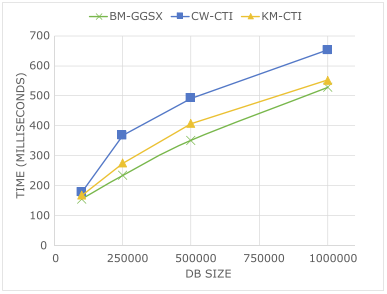
(c) 40 Edges Query Filtering Time



(d) 8 Edges Query Total Response Time

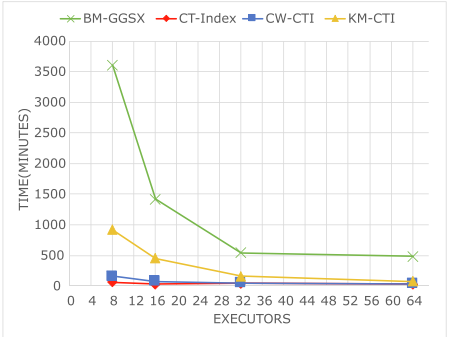


(e) 20 Edges Query Total Response Time

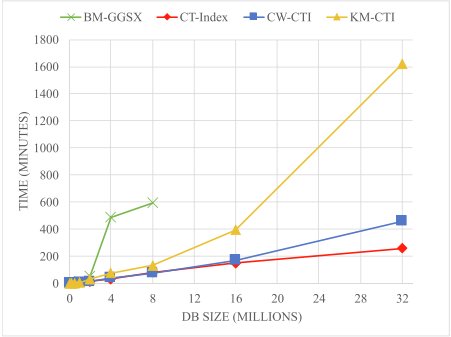


(f) 40 Edges Query Total Response Time

Fig. 6. Average filtering and total query response time in centralized implementations.



(a) 4M DB varying number of executors



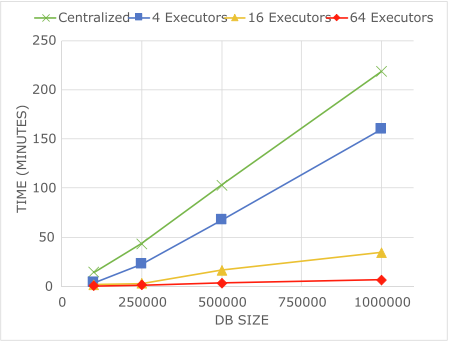
(b) 64 executors varying DB size

Fig. 7. Index Building Time in distributed system varying (a) number of executors and (b) database size.

Fig. 8(a) and (b), respectively for BM-GGSX and KM-CTI. In particular, the centralized implementation of BM-GGSX required more than a whole day to generate the index for 1M graphs, whereas its distributed implementation with 64 executors can do it in only 14 min. Similarly, the time is reduced from 3.5 h to 7 min for KM-CTI.

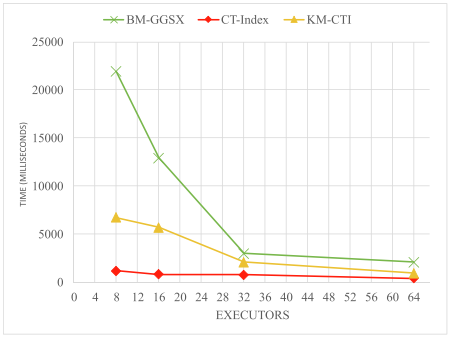
Fig. 9(a) and (b) show the time of the filtering stage for small and large queries, respectively, for a database size of 4M graphs and increasing number of executors. CW-CTI is not shown in the figures due to its low performance (an order of magnitude worse than the other). In general, the response time decreases drastically when the number of executors are increased from 8 to 32. This is reasonable given that the hardware has 16 nodes with 2 processors each, i.e., 32 processors. CT-Index has the best performance for small queries and KM-CTI is slightly better for large ones, specially when the number of executors in low, i.e., when the number of graphs per partition is larger. A similar behaviour is also found for total query time, as it is shown in Fig. 9(c) and (d) for small and large queries, respectively. The impact of the database size in the query response time is shown in Fig. 10. In particular, Fig. 10(a) and (b) show that CT-Index has better filtering time for small queries whereas KM-CTI behaves slightly better for large ones. Besides, it is observed in Fig. 10(c) that CT-Index and KM-CTI have

(a) Index Building Time for BMGGSX

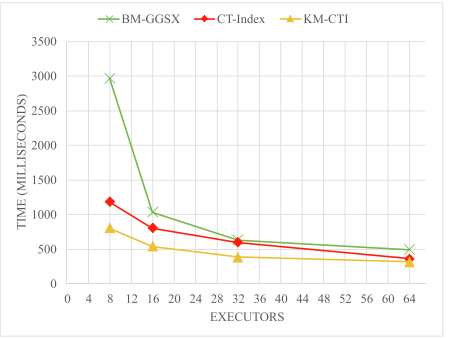


(b) Index Building Time for KM-CTI

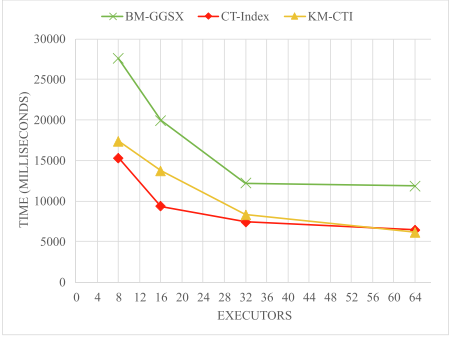
Fig. 8. Comparison of index building time between centralized and distributed implementations for (a) BM-GGSX and (b) KM-CTI.



(a) Filtering Time on 4M DB for small queries



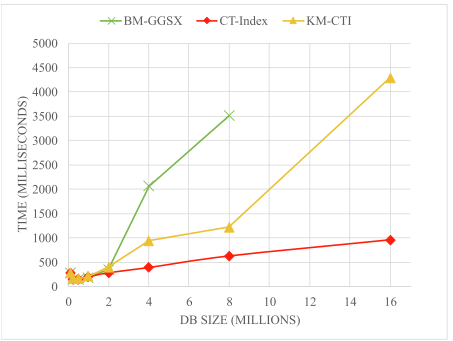
(b) Filtering Time on 4M DB for large queries



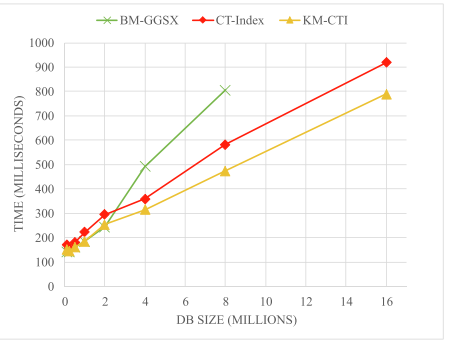
(c) Total Time on 4M DB for small queries

(d) Total Time on 4M DB for large queries

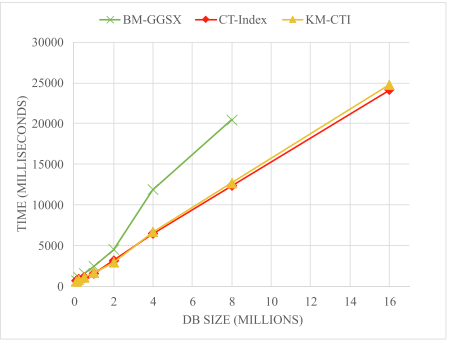
Fig. 9. Average filtering and total response time for distributed implementations (4 M graph database size).



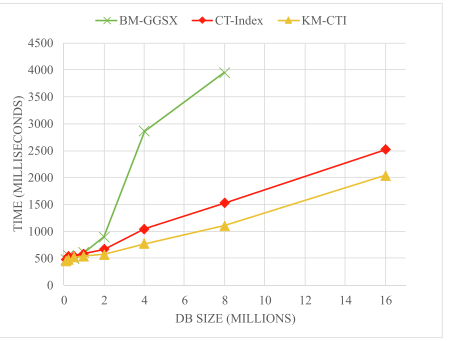
(a) Filtering Time with 64E for small queries



(b) Filtering Time with 64E for large queries

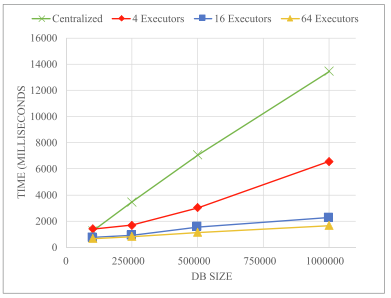


(c) Total Time with 64E for small queries

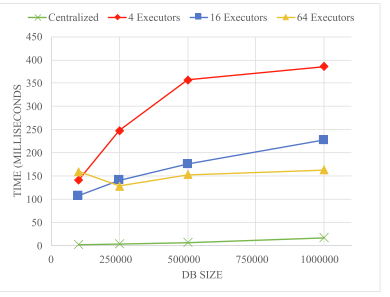


(d) Total Time with 64E for large queries

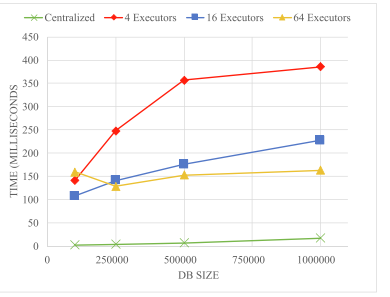
Fig. 10. Average filtering and total response time for distributed implementation (64 executors).



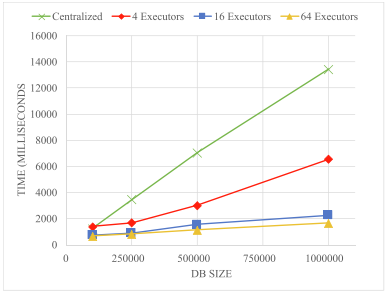
(a)Filtering Time for small size queries



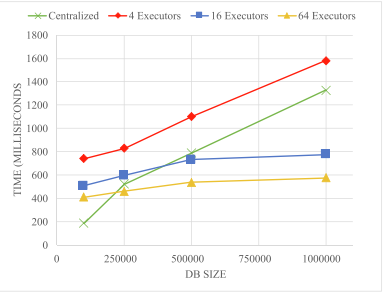
(b)Filtering Time for medium size queries



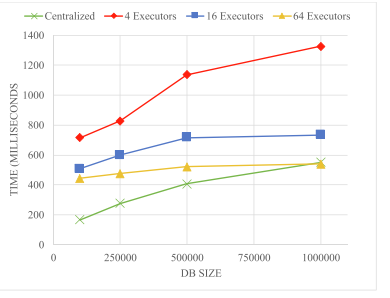
(c)Filtering Time for large size queries



(d)Total Time for small size queries



(e)Total Time for medium size queries



(f) Total Time for large size queries

Fig. 11. Comparison of query filtering and total response time between centralized and distributed implementations for KM-CTI method.

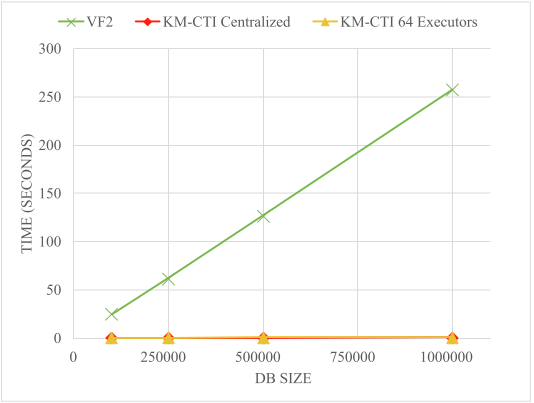


Fig. 12. Illustration of the significance of the use of indexing with respect to the simple parallel evaluation of a subgraph isomorphism algorithm.

similar performance for small queries, i.e., as it was expected, the filtering time does not have a great impact in the total response time for small queries. On the other hand, for large queries (Fig. 10(c)) KM-CTI has better performance than CT-Index, as it was the case for filtering time.

Apart from showing which method behaves better under which configurations and database sizes, this evaluation shows also when it is better to use a centralized or distributed implementation. Thus, Fig. 11(a)–(c) compare the centralized implementation of KM-CTI with the distributed one using 4, 16 and 64 executors, respectively, in terms of query filtering time. The distributed implementation leverages the available hardware during the evaluation of low selective small queries, whereas the centralized implementation has better performance for highly selective medium and large queries. If total response query time is considered (see Fig. 11(d)–(f)), again the distributed implementation is clearly better for small queries, but now for medium and large queries, the parallel execution of the verification stage makes the distributed implementation also better with large database sizes (more than 500k graphs for medium size queries and more than 1 M graphs for large queries).

As a final experiment, one of the proposed indexing techniques (KM-CTI) was compared with the direct parallel evaluation of the VF2 subgraph isomorphism algorithm. The results are shown in Fig. 12. Clearly, indexing the database and performing the filtering stage enables query response times orders of magnitude faster, even for small database sizes.

9. Conclusions

Three new FTV solutions for subgraph search on large databases of small graphs are proposed in this work. The new solutions are based on already existing state of the art methods, GGSX and CT-Index. BM-GGSX uses very large compressed bitmaps to improve the performance of intermediate candidate set intersections. The method is fast for medium and large queries, but it suffers from large index size and slow index building. CW-CTI leverages large compressed bitmaps for the column-wise storage of graph fingerprints. It shows good performance for small queries and also good performance in terms of index size and index building time. Finally, the KM-CTI uses the K-Means clustering method to generate binary trees of graph fingerprints, which are used to speed-up the filtering stage, achieving very good performance for medium and large highly selective queries. Its main drawback is its overhead in both index size and index building time, with respect to the original CT-Index. The combination of CW-CTI for small queries and KM-CTI for medium and large ones is a good global approach for subgraph search query processing. Besides, the filtering stage of the techniques enables retrieving approximate results in interactive time (few milliseconds) for database sizes of up to one million graphs.

A generic solution for the distributed implementation of all the above methods was also developed on top of the large scale data processing engine Apache Spark. To achieve this, binary serializations of the above indexing techniques were defined and search algorithms that work directly on those serializations were implemented. The distributed implementations reduce significantly the index building time overhead of KM-CTI and BM-GGSX. CT-Index has a good compromise between index building time and query response time if many executors are used. KM-CTI is better for small or crowded clusters, where the number of executors has to be low and consequently the number of graphs per partition must be large. Again, the filtering stage enables the retrieval of approximate query responses with interactive response times (below 1 s), even for databases of more than 16 million graphs.

In general, as it was expected, the centralized implementations have good performance with highly selective large queries over small databases, however, the parallel implementations leverage powerful clusters for either small low selective queries or large database sizes. Future work includes i) the exploration of similar solutions for datasets of just one very large graph as those of the semantic web and ii) the combination with indexes on other fields, such as text indexes, to achieve efficient hybrid solutions for composed queries.

CRediT authorship contribution statement

David Luaces: Conceptualization, Investigation, Software, Validation, Writing - original draft, Writing - original draft. **José R.R. Viqueira:** Supervision, Project administration, Funding acquisition, Conceptualization, Investigation, Writing - review & editing. **José M. Cotos:** Project administration, Funding acquisition, Writing - review & editing. **Julián C. Flores:** Project administration, Funding acquisition, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been co-funded by the Ministerio de Economía y Competitividad of the Spanish government, and by Mestrelab Research S.L. through the project NEXTCHROM (RTC-2015-3812-2) of the call Retos-Colaboración of the program Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad. The authors wish to thank the financial support provided by Xunta de Galicia under the Project ED431B 2018/28.

References

[1] H.-C. Ehrlich, A. Volkamer, M. Rarey, Searching for substructures in fragment spaces, Journal of Chemical Information and Modeling 52 (12) (2012) 3181–3189, pMID: 23205736. doi:10.1021/ci300283a. .
[2] H.-C. Ehrlich, M. Rarey, Systematic benchmark of substructure search in molecular graphs – from ullmann to vf2, Journal of Cheminformatics 4 (1) (2012), https://doi.org/10.1186/1758-2946-4-13.
[3] X. Yan, P. Yu, J. Han, Graph indexing: A frequent structure-based approach, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004, pp. 335–346.
[4] S. Zhang, M. Hu, J. Yang, Treepi: A novel graph indexing method, in: Proceedings – International Conference on Data Engineering, 2007, pp. 966–975. .
[5] J. Cheng, Y. Ke, W. Ng, A. Lu, FG-Index: Towards verification-free query processing on graph databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2007, pp. 857–872.
[6] P. Zhao, J.X. Yu, P.S. Yu, Graph indexing: Tree + delta >= graph, in: 33rd International Conference on Very Large Data Bases, VLDB 2007 – Conference Proceedings, 2007, pp. 938–949. .
[7] L. Zou, L. Chen, J.X. Yu, Y. Lu, A novel spectral coding in a large graph database, in: Advances in Database Technology – EDBT 2008 – 11th International Conference on Extending Database Technology, Proceedings, 2008, pp. 181–192.
[8] K. Klein, N. Kriege, P. Mutzel, Ct-index, Fingerprint-based graph indexing combining cycles and trees, in: 2011 IEEE 27th International Conference on Data Engineering (ICDE), IEEE, 2011, pp. 1115–1126.
[9] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, D. Shasha, Enhancing graph database indexing by suffix tree structure, in: IAPR International Conference on Pattern Recognition in Bioinformatics, Springer, 2010, pp. 195–203.
[10] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, D. Shasha, Grapes: A software for parallel searching on biological graphs targeting multi-core architectures, PLoS One 8 (10) (2013).
[11] C.-H. Lee, C.-W. Chung, Efficient search in graph databases using cross filtering, Information Sciences 286 (2014) 1–18, https://doi.org/10.1016/j.ins.2014.06.047.
[12] D. Luaces, J.R. Viqueira, T.F. Pena, J.M. Cotos, Leveraging bitmap indexing for subgraph searching, in: 22nd International Conference on Extending Database Technology (EDBT), OpenProceedings.org, 2019, pp. 49–60. doi:10.5441/002/edbt.2019.06. .
[13] W.-S. Han, J. Lee, M.-D. Pham, J. Yu, igraph, A framework for comparisons of disk based graph indexing techniques, Proceedings of the VLDB Endowment 3 (1) (2010) 449–459.
[14] F. Katsarou, N. Ntarmos, P. Triantafillou, Performance and scalability of indexed subgraph query processing methods, Proceedings of the VLDB Endowment 8 (12) (2015) 1566–1577.
[15] J.R. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM 23 (1) (1976) 31–42.
[16] L. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub) graph isomorphism algorithm for matching large graphs, IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (10) (2004) 1367–1372.
[17] H. Shang, Y. Zhang, X. Lin, J.X. Yu, Taming verification hardness: An efficient algorithm for testing subgraph isomorphism, Proceedings of the VLDB Endowment 1 (1) (2008) 364–375.
[18] H. He, A.K. Singh, Graphs-at-a-time, Query language and access methods for graph databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008, pp. 405–417.
[19] S. Zhang, S. Li, J. Yang, Gaddi, Distance index based subgraph matching in biological networks, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology EDBT'09, 2009, pp. 192–203.
[20] P. Zhao, J. Han, On graph query optimization in large networks, Proceedings of the VLDB Endowment 3 (1) (2010) 340–351.
[21] J. Lee, W.-S. Han, R. Kasperovics, J.-H. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases, in: Proceedings of the VLDB Endowment, vol. 6, 2012, pp. 133–144. .
[22] W.-S. Han, J. Lee, J.-H. Lee, Turboiso, Towards ultrafast and robust subgraph isomorphism search in large graph databases, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 337–348.
[23] W. Zheng, L. Zou, X. Lian, H. Zhang, W. Wang, D. Zhao, Sqbc: An efficient subgraph matching method over large and dense graphs, Information Sciences 261 (2014) 116–131, https://doi.org/10.1016/j.ins.2013.10.003.
[24] X. Ren, J. Wang, Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs, Proceedings of the VLDB Endowment 8 (5) (2015) 617–628.
[25] J. Wang, X. Ren, S. Anirban, X.-W. Wu, Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs, Information Sciences 482 (2019) 363–373, https://doi.org/10.1016/j.ins.2019.01.036.
[26] F. Bi, L. Chang, X. Lin, L. Qin, W. Zhang, Efficient subgraph matching by postponing cartesian products, in: Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, ACM, 2016, pp. 1199–1214.
[27] M. Han, H. Kim, G. Gu, K. Park, W.-S. Han, Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, ACM, New York, NY, USA, 2019, pp. 1429–1446. doi:10.1145/3299869.3319880. .

[28] B. Bhattarai, H. Liu, H. Howie Huang, Ceci: Compact embedding cluster index for scalable subgraph matching, 2019, pp. 1447–1462. doi:10.1145/3299869.3300086. .
[29] X. Ren, J. Wang, W.-S. Han, J.X. Yu, Fast and robust distributed subgraph enumeration, Proceedings of the VLDB Endowment 12 (11) (2019) 1344–1356, https://doi.org/10.14778/3342263.3342272.
[30] X. Jin, L. Lai, Mpmatch: A multi-core parallel subgraph matching algorithm, 2019, pp. 241–248. doi:10.1109/ICDEW.2019.000-6. .
[31] J. Wang, N. Ntarmos, P. Triantafillou, Graphcache: A caching system for graph queries, in: Advances in Database Technology – EDBT, vol. 2017-March, 2017, pp. 13–24. .
[32] J. Wang, N. Ntarmos, P. Triantafillou, Indexing query graphs to speedup graph query processing, in: Advances in Database Technology – EDBT, vol. 2016-March, 2016, pp. 41–52. .
[33] F. Katsarou, N. Ntarmos, P. Triantafillou, Subgraph querying with parallel use of query rewritings and alternative algorithms, in: Advances in Database Technology – EDBT, vol. 2017-March, 2017, pp. 25–36. .
[34] F. Katsarou, N. Ntarmos, P. Triantafillou, Hybrid algorithms for subgraph pattern queries in graph databases, vol. 2018-January, 2017, pp. 656–665. doi:10.1109/BigData.2017.8257981. .
[35] S. Sun, Q. Luo, Scaling up subgraph query processing with efficient subgraph matching, vol. 2019-April, 2019, pp. 220–231. doi:10.1109/ICDE.2019.00028. .
[36] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426, https://doi.org/10.1145/362686.362692.
[37] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Communications of the ACM 59 (11) (2016) 56–65, https://doi.org/10.1145/2934664.
[38] D. Lemire, O. Kaser, K. Aouiche, Sorting improves word-aligned bitmap indexes, Data and Knowledge Engineering 69 (1) (2010) 3–28.
[39] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, vol. 1: Statistics, University of California Press, Berkeley, 1967, pp. 281–297. .
[40] E. Forgy, Cluster analysis of multivariate data: efficiency versus interpretability of classifications, Biometrics 21 (1965) 768–780.
[41] R.W. Hamming, Error detecting and error correcting codes, Bell System Technical Journal 29 (2) (1950) 147–160.